

A Crash Course in Supercomputing

NATIONAL CENTER
FOR COMPUTATIONAL SCIENCES



presented by

Rebecca Hartman-Baker
Oak Ridge National Laboratory

hartmanbakrj@ornl.gov

© 2004-2008 Rebecca Hartman-Baker. Reproduction permitted for
non-commercial, educational use only.

Oak Ridge National Laboratory
U.S. Department of Energy

Outline

- I. Parallelism
- II. Supercomputer Architecture
- III. Makefiles and Batch Scripts
- IV. MPI
- V. OpenMP
- VI. Debugging and Performance Evaluation
- VII. Example: Computing π in Parallel

NATIONAL CENTER
FOR COMPUTATIONAL SCIENCES

Oak Ridge National Laboratory



U.S. Department of Energy



I. PARALLELISM

Parallel Lines by Blondie. Source: <http://xponentialmusic.org/blogs/885mmm/2007/10/09/403-blondie-hits-1-with-heart-of-glass/>

I. Parallelism

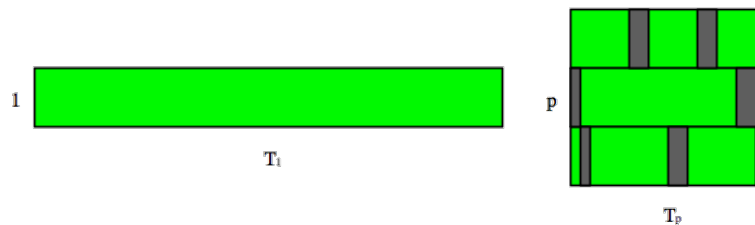
- Concepts of parallelization
- Serial vs. parallel
- Parallelization strategies

Parallelization Concepts

- When performing task, some subtasks depend on one another, while others do not
- Example: Preparing dinner
 - **Salad prep independent of lasagna baking**
 - **Lasagna must be assembled before baking**
- Likewise, in solving scientific problems, some tasks independent of one another

Serial vs. Parallel

- Serial: tasks must be performed in sequence
- Parallel: tasks can be performed independently in any order



Serial vs. Parallel: Example

- Example: Preparing dinner
 - **Serial tasks:** making sauce, assembling lasagna, baking lasagna; washing lettuce, cutting vegetables, assembling salad
 - **Parallel tasks:** making lasagna, making salad, setting table



Serial vs. Parallel: Example

- Could have several chefs, each performing one parallel task
- This is concept behind parallel computing



Parallel Algorithm Design: PCAM

- *Partition*: Decompose problem into fine-grained tasks to maximize potential parallelism
- *Communication*: Determine communication pattern among tasks
- *Agglomeration*: Combine into coarser-grained tasks, if necessary, to reduce communication requirements or other costs
- *Mapping*: Assign tasks to processors, subject to tradeoff between communication cost and concurrency

(taken from *Heath: Parallel Numerical Algorithms*)

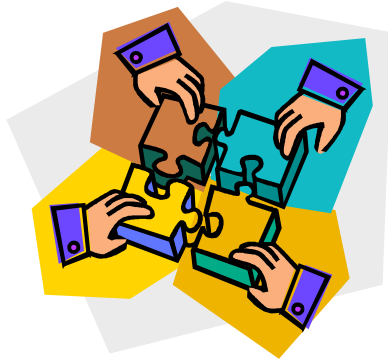
Discussion: Jigsaw Puzzle*

- Suppose we want to do 5000 piece jigsaw puzzle
- Time for one person to complete puzzle: n hours
- How can we decrease wall time to completion?



* Thanks to Henry Neeman

Discussion: Jigsaw Puzzle

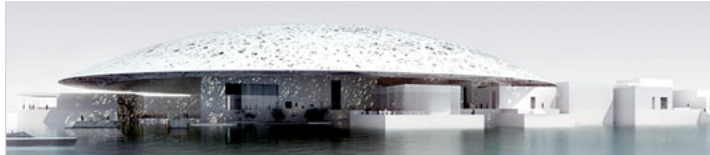


- Add another person at the table
 - Effect on wall time
 - Communication
 - Resource contention
- Add p people at the table
 - Effect on wall time
 - Communication
 - Resource contention

Discussion: Jigsaw Puzzle



- What about: p people, p tables, $5000/p$ pieces each?
- What about: one person works on river, one works on sky, one works on mountain, etc.?



II. ARCHITECTURE

Image: Louvre Abu Dhabi – Abu Dhabi, UAE, designed by Jean Nouvel, from <http://www.inhabitat.com/2008/03/31/jean-nouvel-named-2008-pritzker-architecture-laureate/>

NATIONAL CENTER
FOR COMPUTATIONAL SCIENCES
Oak Ridge National Laboratory

U.S. Department of Energy

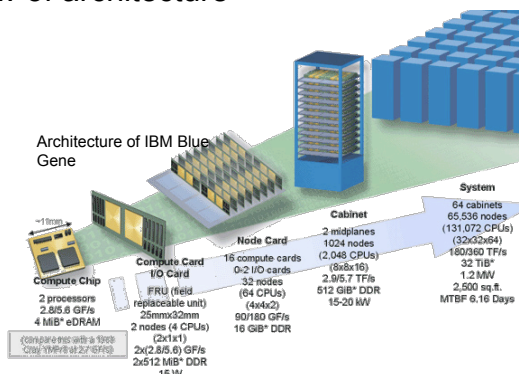
II. Supercomputer Architecture

- What is a supercomputer?
- Conceptual overview of architecture

Cray
(1976)



IBM Blue
Gene
(2005)



NATIONAL CENTER
FOR COMPUTATIONAL SCIENCES
Oak Ridge National Laboratory

U.S. Department of Energy

What Is a Supercomputer?

- “The biggest, fastest computer right this minute.” -- Henry Neeman
- Generally 100-10,000 times more powerful than PC
- This field of study known as *supercomputing*, *high-performance computing (HPC)*, or *scientific computing*
- Scientists use really big computers to solve really hard problems

SMP Architecture

- Massive memory, shared by multiple processors
- Any processor can work on any task, no matter its location in memory
- Ideal for parallelization of sums, loops, etc.

Cluster Architecture

- CPUs on racks, do computations (fast)
- Communicate through myrinet connections (slow)
- Want to write programs that divide computations evenly but minimize communication

State-of-the-Art Architectures

- Today, hybrid architectures gaining acceptance
- Multiple {dual, quad}-core nodes, connected to other nodes by (slow) interconnect
- Cores in node share memory (like small SMP machines)
- Machine appears to follow cluster architecture (with multi-core nodes rather than single processors)
- To take advantage of all parallelism, use MPI (cluster) and OpenMP (SMP) hybrid programming



III. MAKEFILES AND BATCH SCRIPTS

Fortune cookie-shaped USB drives available from <http://vavolo.com/freshlybakedusb.asp>

NATIONAL CENTER
FOR COMPUTATIONAL SCIENCES

Oak Ridge National Laboratory



U.S. Department of Energy

Outline

- Makefiles
- Batch Scripts



NATIONAL CENTER
FOR COMPUTATIONAL SCIENCES

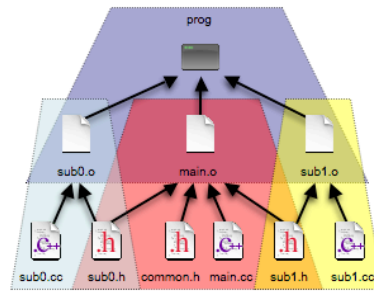
Oak Ridge National Laboratory



U.S. Department of Energy

Makefiles

- Motivation
- Makefile concepts
- Tips
- Resources



```
all: sub0.o sub1.o main.o
    mpicxx sub0.o sub1.o main.o -o prog

sub0.o: sub0.cc sub0.h
    mpicxx -c sub0.cc

sub1.o: sub1.cc sub1.h
    mpicxx -c sub1.cc

main.o: main.cc common.h sub0.h sub1.h
    mpicxx -c main.cc

clean:
    rm *.o prog
```

Motivation

- Easy to compile program if only one file:
`gcc -o program myprog.c`
- (Could be) Easy to compile program if multiple files:
`gcc -c *.c; gcc -o program *.o`
- But what if files in multiple directories? What if using libraries? What if special instructions for certain files?
- Also, what if we made one tiny change in one file, and we had 1000 files in program? We would have to wait for hours for program to compile!

Makefile Concepts

- Makefile: File containing sets of rules for compilation of program(s)
- To use, create file called `Makefile` with these rules, then type `make` (plus target)
- Basic structure of a rule:

```
target ... : dependencies ...  
  
        command  
  
        ...  
  
        ...
```

- Make will manage compilation and recompile only objects that are older than respective source file

Makefile Concepts

- General format of Makefile:
 - First, definitions of variables, e.g.

```
CC          = gcc  
LIB_LIST    = -lm -lpthread  
OBJS        = myprog.o mysub1.o mysub2.o
```
 - Rules, e.g.

```
prog: $(OBJS)  
      $(LINKER) $(OPTFLAGS) -o prog \  
      $(OBJS) $(LIB_DIR) $(LIBS)
```
- In rule, second line (and subsequent lines) starts with tab. *Must* be tab, not spaces!

Sample Makefile (1)

```
CC          = gcc
FC          = g77
CLINKER     = gcc
OPTFLAGS    = -O
INCLUDE_DIR = -I/opt/mpich/include
LIB_DIR     = -L/opt/mpich/lib
LIB_LIST    = -lmpich -lpthread
CFLAGS      = $(OPTFLAGS)
LIBS        = $(LIB_LIST) -lm
# this is a comment
OBJS        = myprog.o mysub1.o mysub2.o \
              mysub3.o mysub4.o
EXEC        = prog
```

Sample Makefile (2)

```
prog: $(OBJS)
      $(CLINKER) $(OPTFLAGS) -o $(EXEC) \
      $(OBJS) $(LIB_DIR) $(LIBS)

clean:
      /bin/rm -f *.o *~ $(EXEC)

.c.o:
      $(CC) $(INCLUDE_DIR) $(CFLAGS) -c $*.c

.f.o:
      $(FC) -c $*.f
```

Tips

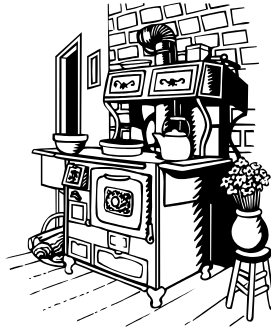
- Error messages from make cryptic; common source of error is using spaces instead of tabs
- `man make` gives good explanation of makefiles
- In above makefile, doing `make clean` removes all object files and gives “clean slate”
- Make will issue message ‘Nothing to be done’ or ‘Target up to date’ if no source files newer than object files

Makefile Resources

- GNU make http://theory.uwinnipeg.ca/gnu/make/make_toc.html
- Make -- a Tutorial <http://www.eng.hawaii.edu/Tutor/Make/>
- Oram, Andrew, and Steve Talbott. *Managing Projects with make*, O'Reilly & Associates, 1991.

Batch Scripts

- Batch System and Scheduling
- Concepts
- Useful commands
- Further help



Batch System and Scheduling

- Supercomputer: powerful computer consisting of many interlinked CPUs
- Users competing for computational resources
- How to launch and schedule jobs fairly?
- Job can run without user presence
- Must not allow one user to hog resources

Batch System

- Batch system accepts input jobs into queue and launches them when resources available
- Many machines use batch system PBS (*P*ortable *B*atch System)
- PBS developed for NASA in 1990s

Scheduler

- Scheduler decides when jobs can be run based on scheduling policies, e.g. user priority, length of job, number of nodes requested, length of time in queue
- Many machines use Maui Scheduler
- Maui Scheduler extensively developed, supported by large segment of computation community including U.S. Dept. of Energy, NCSA



(source: www.the-hawaii-vacation-guide.com)

Concepts

- Limits for walltime and number of processors, so if request exceeds limits, job automatically rejected
- Scheduler rules complicated, but generally, “smaller” jobs run first
- Size of job is function of number of processors and estimated time
- You provide info about number of processors you want and estimate of time job will run

Concepts

- Strategies:
 - **Like inverse of “The Price Is Right,” give lowest estimate possible, without going under true time needed (always good strategy)**
 - **Use fewer processors if possible (usually good strategy)**
- If you reach end of estimated time, PBS will terminate your job!
- Write script that tells PBS what to do when job is launched

Concepts

- Shell Script format:
 - **First, a line invoking the scripting language:**
`#!/bin/csh`
 - **Next, embedded PBS commands, e.g.**
`#PBS -l walltime=00:10:00,nodes=2:ppn=2`
`#PBS -q workq`
(the shell script interprets these as comments, but PBS understands they are PBS commands)
 - **Then, environment variable initialization, e.g.**
`setenv MYMAINDIR /home/hqi/hello` (sets variable MYMAINDIR to /home/hqi/hello)
`setenv PROG $MYMAINDIR/prog` (sets PROG to /home/hqi/hello/prog)

Concepts

- Shell script format (continued):
 - **Then, shell script and regular Linux commands, e.g.,**
`if (-e $OUTF) mv $OUTF $OUTF.old`
(meaning that if file called \$OUTF exists, rename it to \$OUTF.old)
 - **Finally, run job:**
`mpirun -np $NP $PROG < $INFILE > $OUTF`
- To launch job:
 - **Make script executable*:** `chmod u+x myscript`
 - `qsub myscript`

*Not necessary on some systems

Useful Commands (PBS)

- `#PBS -l walltime=hh:mm:ss,nodes=n:ppn=p`
This tells PBS how much walltime you request (where `hh:mm:ss` replaced by appropriate number of hours, minutes, and seconds), how many *dual processor* nodes you want (replace `n` with appropriate number), *and how many processors per node (1 or 2)*
- `#PBS -q workq` Which queue to use (in this case, queue called `workq`)
- `#PBS -v` Export all environment variables to batch job (good practice to do this)
- `#PBS -m be` Sends you e-mail at beginning and end of job

Useful Commands (Shell Scripting)

- `set echo` Print out commands as they are executed (useful for debugging script)
- `setenv A B` Sets environment variable `A` to `B`
- `$A` value of `A`
- `mpirun -np $NP $PROG < $INPUT > $OUTPUT`
`mpirun` (sometimes `mpiexec`, or on proprietary systems, `aprun`, `poe`, etc.) is executable that launches parallel jobs on multiple processors; `-np` is flag indicating number of processors used in run
***NOTE: some implementations do not require input redirection (<)**

Nice Job Script for Institutional Cluster (1)

```
#PBS -S /bin/bash
#PBS -V
#PBS -j oe
#PBS -m ae
#PBS -M hartmanbakrj@ornl.gov
#PBS -N loadbal
#PBS -l walltime=00:10:00,nodes=2:ppn=2
#PBS -q workq
echo "Current working directory is `pwd`"
echo "Node file: $PBS_NODEFILE : "
echo "-----"
cat $PBS_NODEFILE
echo "-----"
NUM_PROCS=`/bin/awk 'END {printNR}' $PBS_NODEFILE`
```

Nice Job Script for Institutional Cluster (2)

```
EXEC=${PBS_O_WORKDIR}/myprog
INPUT_FILE=${PBS_O_WORKDIR}/prog_input.dat
echo "-----"
cat $INPUT_FILE
echo "-----"
echo "Running on $NUM_PROCS processors."
echo "-----"
echo "Starting run at: `date`"
echo "-----"
mpiexec $EXEC $INPUT_FILE
echo "-----"
echo "Ending run at: `date`"
```

Further Help

- NCSA Cobalt Documentation: Running Jobs <http://www.ncsa.uiuc.edu/UserInfo/Resources/Hardware/SGIAItix/Doc/Jobs.html>
- The C Shell tutorial <http://www.eng.hawaii.edu/Tutor/csh.html>
- DuBois, Paul. *Using csh & tcsh*, O'Reilly & Associates, 1995.
- Newham, Cameron and Bill Rosenblatt. *Learning the bash Shell*, O'Reilly & Associates, 1998.

Bibliography/Resources

- About OpenPBS <http://www.openpbs.org/about.html>
- Maui Scheduler <http://www.supercluster.org/maui/>



IV. MPI

MPI also stands for Max Planck Institute for Psycholinguistics. Source: <http://www.mpi.nl/WhatWeDo/institute-pictures/building>

IV. MPI

- Introduction to MPI
- Parallel programming concepts
- The Six Necessary MPI Commands
- Example program

Introduction to MPI

- Stands for *Message Passing Interface*
- Industry standard for parallel programming (200+ page document)
- MPI implemented by many vendors; open source implementations available too
 - **ChaMPIon-PRO, IBM, HP, Cray vendor implementations**
 - **MPICH, LAM-MPI, OpenMPI (open source)**
- MPI function library is used in writing C, C++, or Fortran programs in HPC
- MPI-1 vs. MPI-2: MPI-2 has additional advanced functionality and C++ bindings, but everything learned today applies to both standards

Parallelization Concepts

- Two primary programming paradigms:
 - **SPMD (single program, multiple data)**
 - **MPMD (multiple programs, multiple data)**
- MPI can be used for either paradigm

SPMD vs. MPMD

- SPMD: Write single program that will perform same operation on multiple sets of data
 - **Multiple chefs baking many lasagnas**
 - **Rendering different frames of movie**
- MPMD: Write different programs to perform different operations on multiple sets of data
 - **Multiple chefs preparing four-course dinner**
 - **Rendering different parts of movie frame**
- Can also write hybrid program in which some processes perform same task

The Six Necessary MPI Commands

- `int MPI_Init(int *argc, char **argv)`
- `int MPI_Finalize(void)`
- `int MPI_Comm_size(MPI_Comm comm, int *size)`
- `int MPI_Comm_rank(MPI_Comm comm, int *rank)`
- `int MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)`
- `int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)`

Initiation and Termination

- `MPI_Init(int *argc, char **argv)` initiates MPI
 - Place in body of code after variable declarations and before any MPI commands
- `MPI_Finalize(void)` shuts down MPI
 - Place near end of code, after last MPI command

Environmental Inquiry

- `MPI_Comm_size(MPI_Comm comm, int *size)`
 - Find out number of processes
 - Allows flexibility in number of processes used in program
- `MPI_Comm_rank(MPI_Comm comm, int *rank)`
 - Find out identifier of current process
 - $0 \leq \text{rank} \leq \text{size}-1$

Message Passing: Send

- `MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)`
 - Send message of length `count` bytes and datatype `datatype` contained in `buf` with tag `tag` to process number `dest` in communicator `comm`
 - E.g. `MPI_Send(&x, 1, MPI_DOUBLE, manager, me, MPI_COMM_WORLD)`

Message Passing: Receive

- `MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)`
 - Receive message of length `count` bytes and datatype `datatype` with tag `tag` in buffer `buf` from process number `source` in communicator `comm` and record status `status`
 - E.g. `MPI_Recv(&x, 1, MPI_DOUBLE, source, source, MPI_COMM_WORLD, &status)`

Message Passing

- **WARNING!** Both standard send and receive functions are *blocking*
- `MPI_Recv` returns only after receive buffer contains requested message
- `MPI_Send` may or may not block until message received (usually blocks)
- Must watch out for deadlock

Deadlocking Example (Always)

```
#include <mpi.h>
#include <stdio.h>

int main(int argc, char **argv) {
    int me, np, q, sendto;
    MPI_Status status;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &np);
    MPI_Comm_rank(MPI_COMM_WORLD, &me);
    if (np%2==1) return 0;
    if (me%2==1) {sendto = me-1;}
    else {sendto = me+1;}
    MPI_Recv(&q, 1, MPI_INT, sendto, sendto, MPI_COMM_WORLD, &status);
    MPI_Send(&me, 1, MPI_INT, sendto, me, MPI_COMM_WORLD);
    printf("Sent %d to proc %d, received %d from proc %d\n", me,
           sendto, q, sendto);
    MPI_Finalize();
    return 0;
}
```

Deadlocking Example (Sometimes)

```
#include <mpi.h>
#include <stdio.h>

int main(int argc, char **argv) {
    int me, np, q, sendto;
    MPI_Status status;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &np);
    MPI_Comm_rank(MPI_COMM_WORLD, &me);
    if (np%2==1) return 0;
    if (me%2==1) {sendto = me-1;}
    else {sendto = me+1;}
    MPI_Send(&me, 1, MPI_INT, sendto, me, MPI_COMM_WORLD);
    MPI_Recv(&q, 1, MPI_INT, sendto, sendto, MPI_COMM_WORLD, &status);
    printf("Sent %d to proc %d, received %d from proc %d\n", me,
        sendto, q, sendto);
    MPI_Finalize();
    return 0;
}
```

Deadlocking Example (Safe)

```
#include <mpi.h>
#include <stdio.h>

int main(int argc, char **argv) {
    int me, np, q, sendto;
    MPI_Status status;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &np);
    MPI_Comm_rank(MPI_COMM_WORLD, &me);
    if (np%2==1) return 0;
    if (me%2==1) {sendto = me-1;}
    else {sendto = me+1;}
    if (me%2 == 0) {
        MPI_Send(&me, 1, MPI_INT, sendto, me, MPI_COMM_WORLD);
        MPI_Recv(&q, 1, MPI_INT, sendto, sendto, MPI_COMM_WORLD, &status);
    } else {
        MPI_Recv(&q, 1, MPI_INT, sendto, sendto, MPI_COMM_WORLD, &status);
        MPI_Send(&me, 1, MPI_INT, sendto, me, MPI_COMM_WORLD);
    }
    printf("Sent %d to proc %d, received %d from proc %d\n", me, sendto, q, sendto);
    MPI_Finalize();
    return 0;
}
```

Explanation: Always Deadlock Example

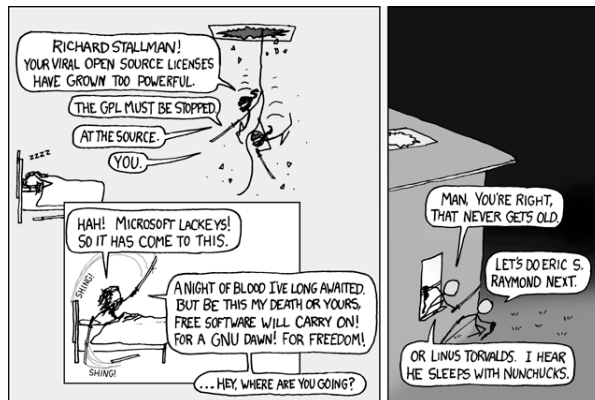
- Logically incorrect
- Deadlock caused by blocking `MPI_Recv`
- All processes wait for corresponding `MPI_Sends` to begin, which never happens

Explanation: Sometimes Deadlock Example

- Logically correct
- Deadlock could be caused by `MPI_Sends` competing for buffer space
- Unsafe because depends on system resources
- Solutions:
 - **Reorder sends and receives, like safe example, having evens send first and odds send second**
 - **Use non-blocking sends and receives or other advanced functions from MPI library (beyond scope of this tutorial)**

Bibliography/Resources: MPI

- Snir, Marc, Steve W. Otto, Steven Huss-Lederman, David W. Walker, and Jack Dongarra. (1996) *MPI: The Complete Reference*. Cambridge, MA: MIT Press. (also available at <http://www.netlib.org/utk/papers/mpi-book/mpi-book.html>)
- MPICH Documentation <http://www-unix.mcs.anl.gov/mpi/mpich/>
- C, C++, and FORTRAN bindings for MPI-1.2 <http://www.lam-mpi.org/tutorials/bindings/>



V. OPENMP

Source: <http://xkcd.com/225/>

V. OpenMP

- About OpenMP
- OpenMP Directives
 - **Parallel**
 - **Loop**
 - **Sections**
 - **Synchronization**
- Data Scope
- Runtime Library Routines
- OpenMP Environment Variables
- Running Applications with OpenMP

About OpenMP

- Industry-standard shared memory programming model
- Developed in 1997
- OpenMP Architecture Review Board (ARB) determines additions and updates to standard

Advantages to OpenMP

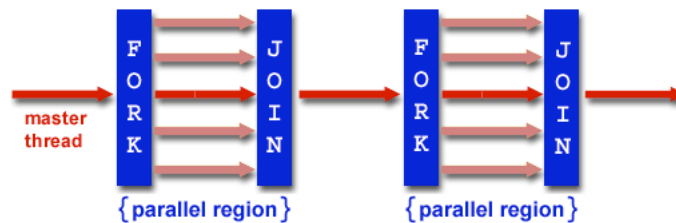
- Parallelize small parts of application, one at a time (beginning with most time-critical parts)
- Can express simple or complex algorithms
- Code size grows only modestly
- Expression of parallelism flows clearly, so code is easy to read
- Single source code for OpenMP and non-OpenMP – non-OpenMP compilers simply ignore OMP directives

OpenMP Programming Model

- Application Programmer Interface (API) is combination of
 - **Directives**
 - **Runtime library routines**
 - **Environment variables**
- API falls into three categories
 - **Expression of parallelism (flow control)**
 - **Data sharing among threads (communication)**
 - **Synchronization (coordination or interaction)**

Parallelism

- Shared memory, thread-based parallelism
- Explicit parallelism (parallel regions)
- Fork/join model



Source: <https://computing.llnl.gov/tutorials/openMP/>

OpenMP Directives: Parallel

- A block of code executed by multiple threads
- Syntax:

```
#pragma omp parallel private(list) \
    shared (list)
{
    /* parallel section */
}
```

Simple Example

```
#include <stdio.h>
#include <omp.h>
int main (int argc, char *argv[]) {
    int tid;
    printf("Hello world from threads:\n");
    #pragma omp parallel private(tid)
    {
        tid = omp_get_thread_num();
        printf("<%d>\n", tid);
    }
    printf("I am sequential now\n");
    return 0;
}
```

Output (Simple Example)

Output 1

Hello world from
threads:

<0>

<1>

<2>

<3>

<4>

I am sequential now

Output 2

Hello world from
threads:

<1>

<2>

<0>

<4>

<3>

I am sequential now

Order of execution is scheduled by OS!!!!!!

OpenMP Directives: Loop

- Iterations of the loop following the directive are executed in parallel

- Syntax:

```
#pragma omp for schedule(type [,chunk]) \  
private(list) shared(list) nowait  
{  
    /* for loop */  
}  
- type = {static, dynamic, guided, runtime}  
- If nowait specified, threads do not synchronize at  
  end of loop
```

OpenMP Directives: Loop Scheduling

- Default scheduling determined by implementation
- Static
 - ID of thread performing particular iteration is function of iteration number and number of threads
 - Statically assigned at beginning of loop
 - Load imbalance may be issue if iterations have different amounts of work
- Dynamic
 - Assignment of threads determined at runtime (round robin)
 - Each thread gets more work after completing current work
 - Load balance is possible

Loop: Simple Example

```
#include <omp.h>
#define CHUNKSIZE 100
#define N      1000
int main () {
    int i, chunk;
    float a[N], b[N], c[N];
    /* Some initializations */
    for (i=0; i < N; i++)
        a[i] = b[i] = i * 1.0;
    chunk = CHUNKSIZE;
    #pragma omp parallel shared(a,b,c,chunk) private(i)
    {
        #pragma omp for schedule(dynamic,chunk) nowait
        for (i=0; i < N; i++)
            c[i] = a[i] + b[i];
    } /* end of parallel section */
    return 0;
}
```

OpenMP Directives: Sections

- Non-iterative work-sharing construct
- Divide enclosed sections of code among threads
- Section directives nested within sections directive
- Syntax

```
#pragma omp sections
{
    #pragma omp section
    /* first section */
    #pragma omp section
    /* next section */
}
```

Sections: Simple Example

```
#include <omp.h>
#define N 1000
int main () {
    int i;
    float a[N], b[N], c[N],
    d[N];
    /* Some initializations
    */
    for (i=0; i < N; i++) {
        a[i] = i * 1.5;
        b[i] = i + 22.35;
    }
}
```

```
#pragma omp parallel shared(a,b,c,d) \
private(i)
{
    #pragma omp sections nowait
    {
        #pragma omp section
        for (i=0; i < N; i++)
            c[i] = a[i] + b[i];
        #pragma omp section
        for (i=0; i < N; i++)
            d[i] = a[i] * b[i];
    } /* end of sections */
} /* end of parallel section */
return 0;
}
```

OpenMP Directives: Synchronization

- Sometimes, need to make sure threads execute regions of code in proper order
 - **Maybe one part depends on another part being completed**
 - **Maybe only one thread need execute a section of code**
- Critical
 - **Specifies section of code that must be executed by only one thread at a time**
 - **Syntax**
`#pragma omp critical [name]`
 - **Names are global identifiers – critical regions with same name are treated as same region**

OpenMP Directives: Synchronization

- Barrier
 - Synchronizes all threads: thread reaches barrier and waits until all other threads have reached barrier, then resumes executing code following barrier
 - Syntax

```
#pragma omp barrier
```
 - Sequence of work-sharing and barrier regions encountered must be the same for every thread
- Single
 - Enclosed code is to be executed by only one thread
 - Useful for thread-unsafe sections of code (e.g., I/O)
 - Syntax

```
#pragma omp single
```

Variable Scope

- By default, all variables shared except
 - Certain loop index values – private by default
 - Local variables and value parameters within subroutines called within parallel region – private
 - Variables declared within lexical extent of parallel region – private

Default Scope Example

```
void caller(int *a, int n) {
    int i,j,m=3;
    #pragma omp parallel for
    for (i=0; i<n; i++) {
        int k=m;
        for (j=1; j<=5; j++) {
            callee(&a[i], &k, j);
        }
    }
}

void callee(int *x, int *y, int
z) {
    int ii;
    static int cnt;
    cnt++;
    for (ii=1; ii<z; ii++) {
        *x = *y + z;
    }
}
```

Var	Scope	Comment
a	shared	Declared outside parallel construct
n	shared	same
i	private	Parallel loop index
j	shared	Sequential loop index
m	shared	Declared outside parallel construct
k	private	Automatic variable/parallel region
x	private	Passed by value
*x	shared	(actually a)
y	private	Passed by value
*y	private	(actually k)
z	private	(actually j)
ii	private	Local stack variable in called function
cnt	shared	Declared static (like global)

OpenMP Runtime Library Routines

- `void omp_set_num_threads(int num_threads)`
 - Sets number of threads used in next parallel region
 - Must be called from serial portion of code
- `int omp_get_num_threads()`
 - Returns number of threads currently in team executing parallel region from which it is called
- `int omp_get_thread_num()`
 - Returns rank of thread
 - $0 \leq \text{omp_get_thread_num}() < \text{omp_get_num_threads}()$

OpenMP Environment Variables

- Set environment variables to control execution of parallel code
- OMP_SCHEDULE
 - **Determines how iterations of loops are scheduled**
 - **E.g., `setenv OMP_SCHEDULE "guided, 4"`**
- OMP_NUM_THREADS
 - **Sets maximum number of threads**
 - **E.g., `setenv OMP_NUM_THREADS 4`**

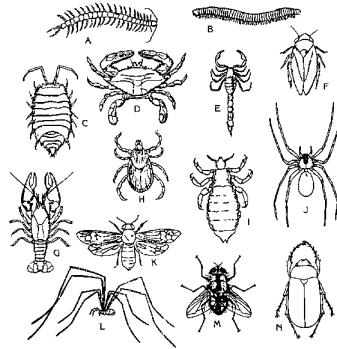
Running Programs with OpenMP Directives

- May need special compiler options (e.g., for PGI compilers, use `-mp=nonuma` flag)
- May need to set environment variables in batch scripts (e.g., on Jaguar, include definition of OMP_NUM_THREADS in script)
- Example: to run on 64 dual-core nodes on Jaguar, add the following to your script:

```
export OMP_NUM_THREADS=2  
aprun -n 128 -N 1 myprog
```


Bibliography/Resources: OpenMP

- Chapman, Barbara, Gabrielle Jost, and Ruud van der Pas. (2008) *Using OpenMP*, Cambridge, MA: MIT Press.
- Kendall, Ricky A. (2007) *Threads R Us*, http://www.nccs.gov/wp-content/training/scaling_workshop_pdfs/threadsRus.pdf
- LLNL OpenMP Tutorial, <https://computing.llnl.gov/tutorials/openMP/>



Source: <http://www.uky.edu/Aq/Entomology/ythfacts/4h/unit1/i&tr.htm>

VI. DEBUGGING AND PERFORMANCE EVALUATION

VI. Debugging and Performance Evaluation

- Common errors in parallel programs
- Debugging tools
- Overview of benchmarking and performance measurements



Common Errors

- Program hangs
 - **Send has no corresponding receive (or vice versa)**
 - **Send/receive pair do not match in source/recipient or tag**
 - **Condition you believe should occur does not occur**
- Segmentation fault
 - **Trying to access memory you are not allowed to access/ memory you should not have been allowed to access has been altered (e.g. array index out-of-bounds, uninitialized pointers, using non-pointer as pointer)**
 - **Trying to access a memory location in a way that is not allowed (e.g. overwrite a read-only location)**

Debugging Tools

- Debugging parallel codes is particularly difficult
- Problem: figuring out what happens on each node
- Solutions:
 - **Print statements, I/O redirection into files belonging to each node**
 - **Debuggers compatible with MPI**

Print Statement Debugging Method

- Each processor dumps print statements to `stdout` or into individual output files, e.g. `log.0001`, `log.0002`, etc.
- Advantage: easy to implement, independent of platform or available resources
- Disadvantage: time-consuming, extraneous information in log files

MPI-Compatible Debuggers

- TotalView
 - **Commercial product, easy-to-use GUI**
 - **Installed on production systems such as Crays, probably not installed on local machines**
- Free debuggers + `mpirun`
 - **Use `mpirun` command and specify your favorite debugger, e.g. `mpirun -dbg=ddd -np 4 ./myprog`**
 - **This option available with MPICH and most other MPI implementations**
 - **Not as “pretty” as TotalView but it gets job done**

Benchmarking and Performance

- Efficiency
- Scalability
- Performance modeling
- Example

Efficiency

- How well does parallel program perform compared to serial program (or parallel program on 1 processor)?

$$E_N = \frac{T_1}{NT_N}$$

- E = efficiency, N = # processors, T_p = time for p processors

Efficiency

- Ideally, $E_N = 1$; realistically, $E_N < 1$.
- Factors influencing efficiency
 - **Load balance** (evenly distribute work for better efficiency)
 - **Concurrency** (minimize idle time on all processors)
 - **Overhead** (minimize work that serial computation would not do, e.g. communication)

Scalability: Speedup

- How well does parallel program take advantage of additional processors?

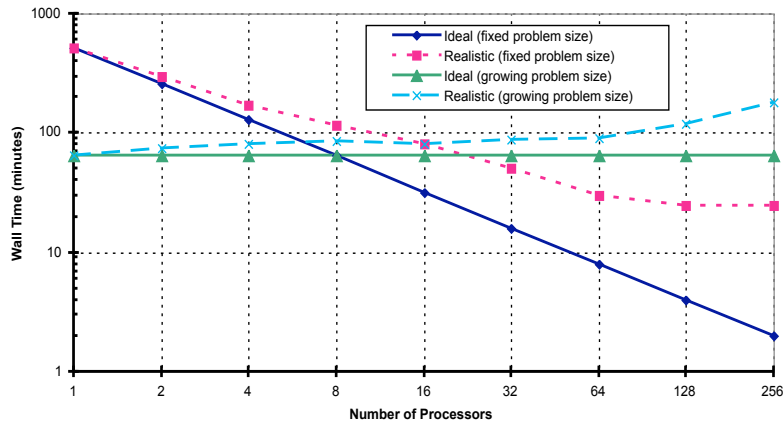
$$S_N = \frac{T_1}{T_N}$$

- S = speedup, N = # processors, T_p = time for p processors

Determining Scalability of Program

- How to measure scalability
 - **Fixed problem size, measure T_N for different N 's**
 - **Increase problem size proportional to N , compare T_N**
- Repeat performance runs at least 3 times for each N (ideally >5 times)
- Plot on log-log graph; slope of line determines scalability

Scalability



Performance Evaluation

- Create performance model

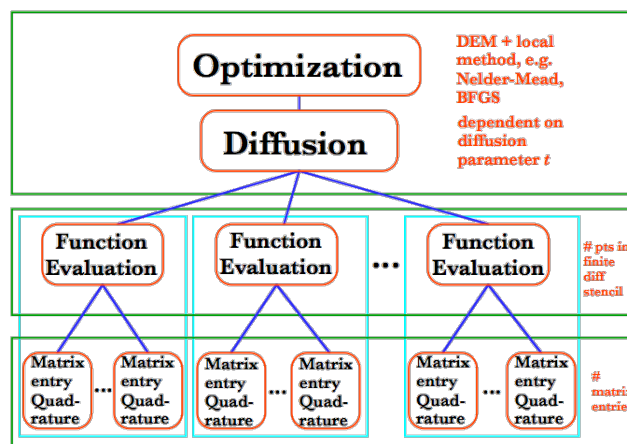
$$T_N = T_N^{\text{communication}} + T_N^{\text{computation}} + T_N^{\text{serial}}$$

- Examine parallel algorithm and figure out which parts fit in each category
- Perform least-squares fit with scalability data

Benchmarking and Performance: Example

- Example of real program: three-tier parallel program from my dissertation
- The problem: Compute diffusion function
 - Compute f matrices, each matrix and each matrix entry independent of all others
 - Perform matrix-vector multiply for each matrix and take norm of result
 - Take weighted average of f results

Example: Schematic Overview of Algorithm



Example: Categorize Algorithm

Communication	Computation	Serial (Idle)
<p><i>Manager</i>: send information about computation to <i>All</i></p> <p><i>Workers</i>: Send matrix entries to <i>Drivers</i></p> <p><i>Drivers</i>: Send results to manager</p>	<p><i>All</i>: Compute matrix entries using quadrature</p> <p><i>Drivers</i>: Compute matrix/vector multiply and norm</p>	<p><i>Manager</i>: Initialize</p> <p>(<i>Worker</i> processes are idle)</p> <p>Compute final function evaluation (<i>All</i> processes except <i>Manager</i> are idle)</p>

Time ↓

NATIONAL CENTER
FOR COMPUTATIONAL SCIENCES

Oak Ridge National Laboratory



U.S. Department of Energy

Example: Performance Evaluation

$$T_N = T_N^{\text{communication}} + T_N^{\text{computation}} + T_N^{\text{serial}}$$

For three-tier algorithm,

$$T_N = (3N + d - 1)t_s + P(N, f)t_{\text{quad}} + t_{\text{init}}$$

- N = # processors
- d = # drivers
- f = stencil size
- $P(N, f)$ = max # entries computed by 1 proc
- t_s = message startup time
- t_{quad} = avg time to compute one entry
- t_{init} = time spent by manager in serial

NATIONAL CENTER
FOR COMPUTATIONAL SCIENCES

Oak Ridge National Laboratory

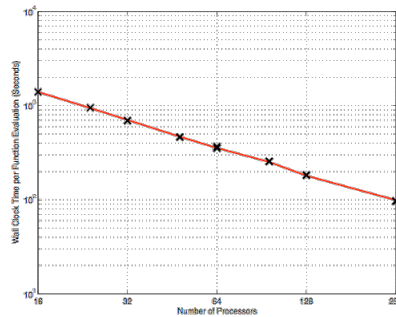


U.S. Department of Energy

Example: Performance Evaluation

- Using least squares solve, we obtain

$$T_N = (3N + d - 1) 3.81077 \times 10^{-3} + P(N, f) 10.3311 + 3.91500 \text{ sec}$$



NATIONAL CENTER
FOR COMPUTATIONAL SCIENCES

Oak Ridge National Laboratory



U.S. Department of Energy

Bibliography/Resources: Programming Concepts and Debugging

- Heath, Michael T. (2006) *Notes for CS554: Parallel Numerical Algorithms*, <http://www.cse.uiuc.edu/cs554/notes/index.html>
- MPI Deadlock and Suggestions <http://www.ncsa.uiuc.edu/UserInfo/Resources/Hardware/CommonDoc/MessPass/MPIDeadlock.html>
- TotalView Tutorial <http://www.llnl.gov/computing/tutorials/totalview/>
- Etnus TotalView page <http://www.etnus.com/>

NATIONAL CENTER
FOR COMPUTATIONAL SCIENCES

Oak Ridge National Laboratory



U.S. Department of Energy

Bibliography/Resources: Benchmarking and Performance

- Heath, Michael T. (2006) *Notes for CS554: Parallel Numerical Algorithms*, <http://www.cse.uiuc.edu/cs554/notes/index.html>
- Hartman-Baker, Rebecca J. (2005) *The Diffusion Equation Method for Global Optimization and Its Application to Magnetotelluric Geoprospecting*, Department of Computer Science, University of Illinois at Urbana-Champaign, <http://www.cs.uiuc.edu/research/techreports.php?report=UIUCDCS-R-2005-2578>



VII. PROGRAMMING PROJECT

Source: http://www.ehow.com/how_2141082_best-berry-pie-ever.html

VII. Programming Project

- Project Description
- Programming Concepts
- Parallelization Strategies

Project Description

- We want to compute π
- One method: method of darts*
- Ratio of area of square to area of inscribed circle proportional to π



*Disclaimer: this is a **TERRIBLE** way to compute π . Don't even think about doing it this way except for the purposes of this project!

Method of Darts

- Imagine dartboard with circle of radius R inscribed in square
- Area of circle $= \pi R^2$
- Area of square $= (2R)^2 = 4R^2$
- $\frac{\text{Area of circle}}{\text{Area of square}} = \frac{\pi R^2}{4R^2} = \frac{\pi}{4}$



Method of Darts

- So, ratio of areas proportional to π
- How to find areas?
 - Suppose we threw darts (completely randomly) at dartboard
 - Could count number of darts landing in circle and total number of darts landing in square
 - Ratio of these numbers gives approximation to ratio of areas
 - Quality of approximation increases with number of darts
- $\pi = 4 \times \frac{\text{\# darts inside circle}}{\text{\# darts thrown}}$



Method of Darts

- Okay, Rebecca, but how in the world do we simulate this experiment on computer?
 - **Decide on length R**
 - **Generate pairs of random numbers (x, y) s.t.**
 $-R \leq x, y \leq R$
 - **If (x, y) within circle (i.e. if $(x^2 + y^2) \leq R^2$), add one to tally for inside circle**
 - **Lastly, find ratio**

Programming Concepts

- Random numbers: in C language, function `int rand (void)` generates “pseudo-random integer in range 0 to `RAND_MAX`”
- `RAND_MAX`: C-language constant denoting maximum random number generated; actual value varies with implementation
- Divide “random” number by maximum random number to get a number between 0 and 1*

*Disclaimer: this is a **TERRIBLE** way to compute a pseudorandom number. Don't even think about doing it this way except for the purposes of this project!

Programming Concepts

- Type cast and coercion:
 - `int a = rand(); double b = a/RAND_MAX;`
 - `b` equals 0
 - `int a = rand(); double b = ((double) a)/((double) RAND_MAX);`
 - `b` equals correct value
 - Type conversion rules:
 - `int/int` → `int`
 - `int/double` → `double`
 - `double/int` → `double`
 - `double/double` → `double`

Programming Concepts

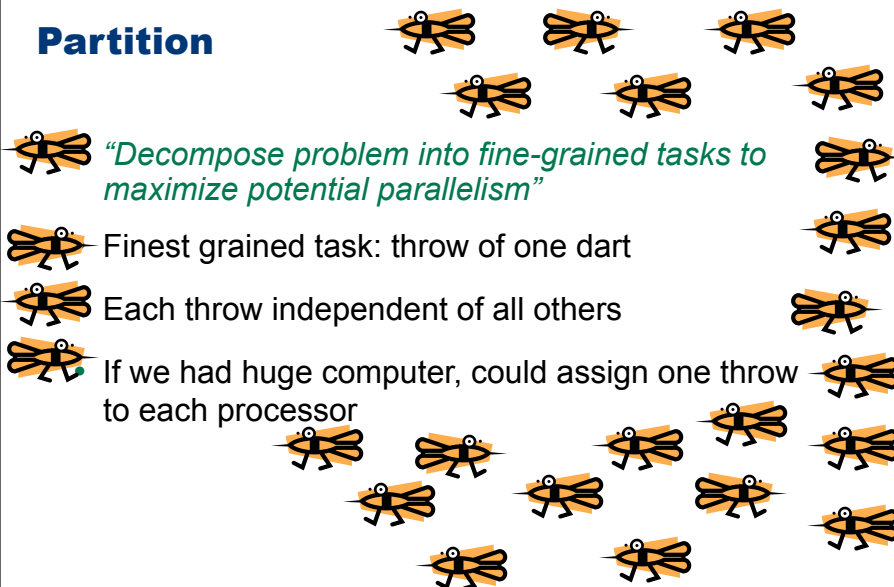
- Numbers generated by `rand()` not really random; same sequence every time
- Change seed for random number generator with `void srand(unsigned int seed)`
- Datatypes:
 - For a lot of darts, need larger datatype than `int` or risk overflow
 - On some computers (varies by platform):

Data Type	Range
<code>int</code>	-32,768 — +32,767
<code>long int</code>	-2,147,483,648 — +2,147,483,648
<code>unsigned long int</code>	0 — +4,294,967,295

Parallelization Strategies

- What tasks independent of each other?
- What tasks must be performed sequentially?
- Using PCAM parallel algorithm design strategy

Partition

The slide features a collection of stylized ants, each with a black body, orange wings, and a small black dot for a head. These ants are scattered across the slide, primarily around the text, to represent individual tasks in a parallel system. There are approximately 25 ants in total.

“Decompose problem into fine-grained tasks to maximize potential parallelism”

Finest grained task: throw of one dart

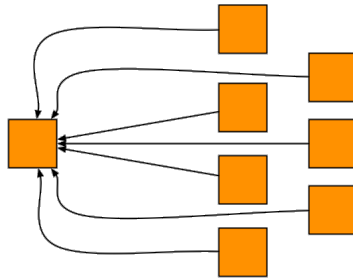
Each throw independent of all others

If we had huge computer, could assign one throw to each processor

Communication

“Determine communication pattern among tasks”

- Each processor throws dart(s) then sends results back to manager process



Agglomeration

“Combine into coarser-grained tasks, if necessary, to reduce communication requirements or other costs”

- To get good value of π , must use millions of darts
- We don't have millions of processors available
- Furthermore, communication between manager and millions of worker processors would be very expensive
- Solution: divide up number of dart throws evenly between processors, so each processor does a share of work

Mapping

“Assign tasks to processors, subject to tradeoff between communication cost and concurrency”

- Assign role of “manager” to processor 0
- Processor 0 will receive tallies from all the other processors, and will compute final value of π
- Every processor, including manager, will perform equal share of dart throws



Bibliography/Resources

- Heath, Michael T. (2006) *Notes for CS554: Parallel Numerical Algorithms*, <http://www.cse.uiuc.edu/cs554/notes/index.html>
- Kernighan, Brian W. and Dennis M. Ritchie. *The C Programming Language*, 2nd ed. Upper Saddle River, NJ: Prentice-Hall, 1988.
- C: The float and double Data Types and the sizeof Operator http://www.iota-six.co.uk/c/b3_float_double_and_sizeof.asp
- C Data types http://www.phim.unibe.ch/comp_doc/c_manual/C/CONCEPT/data_types.html

Appendix: Better Ways to Compute π

- Look it up on the internet, e.g. <http://oldweb.cecm.sfu.ca/projects/ISC/data/pi.html>
- Compute using the BBP (Bailey-Borwein-Plouffe) formula

$$\pi = \sum_{n=0}^{\infty} \left(\frac{4}{8n+1} - \frac{2}{8n+4} - \frac{1}{8n+5} - \frac{1}{8n+6} \right) \left(\frac{1}{16} \right)^n$$

- For less accurate computations, try your programming language's constant, or quadrature or power series expansions

Appendix: Better Ways to Generate Pseudorandom Numbers

- For serial codes
 - Mersenne twister
 - GSL (Gnu Scientific Library), many generators available (including Mersenne twister) <http://www.gnu.org/software/gsl/>
- For parallel codes
 - SPRNG, regarded as leading parallel pseudorandom number generator <http://sprng.cs.fsu.edu/>
 - PPRNG, Bill Cochran's new parallel pseudorandom number generator, supposedly superior to SPRNG <http://runge.cse.uiuc.edu/~wkcochra/pprng/>